TRAVAUX PRATIQUES XIV

Impératif et OCaml

OCaml est un langage fonctionnel [1]... mais pas que. Il permet aussi de faire de la programmation impérative [2].

Remarque : OCaml est pensé en premier lieu comme un langage fonctionnel. Faire de l'impératif lorsqu'un équivalent fonctionnel aussi simple&efficace&lisible est possible est généralement vu comme une mauvaise pratique - et de toute façon, si vous voulez faire avant tout de la programmation impérative il existe des langages pensés pour cela [3].

A Révisions rapides

On rappelle que les fonctions `print_int : int -> unit`, `print_char : char -> unit`, `print_string : string -> unit`, `print_float : char -> unit` permettent de faire ce que leur nom indique. De même pour les équivalents en `read_...`.

1. Écrire une fonction récursive `ligne : int -> char -> unit` telle que `ligne n c` affiche `n` fois le caractère `c`.
2. En déduire une fonction `triangle int -> char -> unit` telle que `pyramide n c` affiche une pyramude à `n` lignes (base en haut) de `c`. Voici par exemple une pyramide de 5 charactères `*` :

```
*****
****
***
**
*
```

B Références, tableaux

B.1 Références

On peut créer l'équivalent d'une variable mutable. Cela s'appelle une référence. Par exemple :

OCaml
```
let x = ref 0
```

Ceci crée x, une référence dont le contenu est 0. Si l'on veut lire au contenu de x, on utilise `!x`. Par exemple :

OCaml
```
let x = ref 0
let _ =
  if !x <= 0 then
    print_string "neg\n"
  else
    print_string "pos\n"
```

⚠ Attention à ne pas confondre `x` la référence, et `!x` le contenu de la référence. Elles n'ont d'ailleurs pas le même type ! Ici `x` est de type `int ref` (« référence dont le contenu est un entier »), et `!x` a le type `int`.

3. Allez taper ces exemples dans l'interpréteur OCaml `utop`. Vérifiez ce que j'affirme sur le type, et vérifiez que l'on ne peut pas remplacer `!x` par `x` dans le `if`.

1. Fonctionnel : penser le calcul comme une suite d'évaluation de fonctions/expressions.
2. impératif : penser le calcul comme une suite de modifications de la mémoire.
3. Comme C.

Bon, tout ça c'est bien beau... mais le but est aussi de pouvoir *modifier* ! On peut changer le contenu de la référence `x` avec l'expression `x := expr` . C'est une expression de type `unit`, qui a l'effet secondaire de modifier le contenu de `x`. Par exemple :

OCaml
```
let y = ref 1
let _ = y := 2 * !y
```

4. Prédisez le nouveau contenu de `y`. Vérifiez dans `utop` .

Et voilà, c'est à peu près tout ce qu'il y a à savoir sur les références. Le reste est une question d'entrainement. Résumons :

Définition 1 : Références

L'expression `ref v` est une référence dont le contenu est `v` . Si `v` est de type `type`, alors `ref v` est de type `type ref`.
Pour lire le contenu d'une référence `r`, on utilise `!r` . Pour modifier le contenu d'une référence `r`, on utilise `r := expr` (cela met la valeur de `expr` dans `r`).

5. Écrire une fonction `incr : int ref -> unit` qui prend en argument une référence d'entier et augmente de 1 son contenu.

6. (Difficile, optionnel) Écrire une fonction qui prend en argument une `int ref` (dont le contenu est positif), et modifie son contenu. Si avant l'appel de la fonction la référence contient `n`, alors après l'appel elle contient le premier entier premier supérieur ou égal à `n`.
Par exemple, si avant l'appel la référence contient 20, alors après l'appel elle contient 23.

Voici un exemple où les références sont très agréables :

OCaml
```
let mystere l =
  let count = ref 0 in
  let rec mystere_loop l = match l with
    | [] -> ()
    | h::t -> let _ = incr count in
              mystere_loop t
  in
  let _ = mystere_loop l in
  !count
```

7. Que fait cette fonction ? Le prédire. Le vérifier en testant dans `utop` .

8. Justifier que `mystere_loop` est terminale récursive, c'est à dire que le seul endroit où elle s'appelle récursivement est à la toute fin d'un de ses appels. *Il s'ensuit que* `mystere` *est en espace constant =)* .

9. Écrire une fonction qui calcule le nombre de termes pairs dans une liste. On utilisera la même architecture de code.

On aurait également pu ne pas utiliser de références, en ajoutant un argument supplémentaire à la fonction auxiliaire qui sert à compter. Mais l'avantage de la référence ici est de garder un code lisible !

B.2 Tableaux

Venons-en à la star incontestée de la prgrammation impérative : les tableaux !

OCaml
```
let t = Array.make 5 1
let _ = print_int t.(3)
```

10. Testez le code précédent dans `utop` . Essayez de modifier `t.(3)` par d'autres valeurs (4 ? 0 ? 7 ? -1 ? 5 ?).

Dans l'exemple précédent, comme vous l'avez peut-être deviné, `t` est le tableau `1 1 1 1 1` .

Définition 2 : make

La fonction `Array.make` prend en argument une longueur `n` et une valeur `v`, et renvoie un tableau à `l` cases dont chaque case est initialisé à `v`.
Si `v` est de type , alors le tableau créé est de type `array` . Plus généralement, un tableau ne peut contenir que des éléments de même type.Si ce type est , alors le tableau est de type .

Définition 3 : Lecture et modification

> Une fois créé, un tableau ne peut pas être redimmensionné.

En plus de lire une valeur comme montrée précédemment, on peut également les modifier (c'est tout l'intérêt d'un tableau !).

> Si `t` est un tableau et `i` un indice de ce tableau, alors `t.(i)` est le contenu de la case d'indice `i` de `t`. C'est une expression qui a le type et la valeur de la case.
> Pour modifier le contenu de la case, on utilise l'expression `t.(i) <- v` . Cela met la valeur `v` dans la case d'indice `i` de `t`. C'est une expression de type `unit`, qui a pour effet secondaire de modifier la case de `t`.

Notez la différence : pour muter une référence, on utilise `:=` . Pour muter une case d'un tableau, on utilise `<-` (et idem pour les types enregistrements mutables que l'on verra plus tard). [4] [5]

11. Créez un tableau nommé `w` à 3 cases contenant toutes `'a'`. Vérifiez.

On peut maintenant modifier la seconde case du tableau pour y mettre un `'b'` ainsi :

OCaml
```
let _ = w.(1) <- 'b'
```

12. Le faire. Vérifier que cela a bien modifié la case annoncée et uniquement la case annoncée.

Il est de plus possible d'accéder au nombre de cases d'un tableau :

Définition 4

> La fonction `Array.length` prend en argument un tableau et renvoie sa longueur (nombre de cases). Elle est de type `'a array -> int` (à un tableau dont le contenu est n'importe quoi, on associe un entier (la longueur)).

Voilà, vous savez presque [6] tout ce qu'il y a à savoir sur les tableaux. Ajoutons un raccourci pratique : de même qu'il y a une notation pour une liste dont on précise le contenu (e.g. `[0;5;6]`), il y a une notation pour un tableau dont on précise le contenu : `[| -2; 5; 3; 0 |]` est le `int array` qui contient… ça se voit. Notez que l'on utilise des points virgules et non des virgules.

13. Écrire une fonction `double_int_array` qui double le contenu de chaque case d'un tableau d'entiers. On utilisera la récursivité pour parcourir les indices.
Indication : si vous devez doubler le contenu des cases d'un tableau entre les indices `i` et `n`, doublez l'élément d'indice `i` puis répétez récursivement entre les indices `i+1` et `n`.

B.3 Mais, et l'imutabilité ?

Les variables en OCaml sont immuables. Mais donc… quid des références ?
Une référence est un pointeur [7]. Le pointeur en lui-même est immuable : on ne peut pas le faire pointer vers un autre endroit de la mémoire. On peut par contre changer la valeur qui est présente à cette adresse.
C'est un peu la même idée pour les tableaux : on ne peut pas les déplacer ni les redimensionner car ils ne peuvent pas bouger en mémoire, mais le contenu de leurs cases peuvent changer.

NB : cette explication permet de comprendre pourquoi on affirme tout de même qu'OCaml est un langage imuable, et de faire du lien avec C. Elle ne correspond toutefois pas forcément au modèle mémoire de OCaml ; aussi faut-il prendre ces histoires de pointeurs avec de grosses pincettes.

C Boucles

C.1 Pour

Pour manipuler des tableaux, les fonctiosn récursives fonctionnent… mais le formalisme des boucles `for` nous manque. C'est pourquoi elles existent !

4. Les concepteurs d'OCaml reconnaissant que cette différence est une erreur, mais argumentent qu'il est trop tard pour la changer.
5. Tout langage assez vieux est *rempli* de choix initiaux maintenant considérés comme des erreurs. C'est normal. Par exemple, C autorisait autrefois les tableaux dont la longueur est une variable mais les a depuis retirés du langage. Similairement, Python a fait il y a quelques années une version 3.0 qui n'était pas compatible avec ses versions antérieures (‼) pour annuler certaines de ses erreurs.
6. Vous noterez que ce TP n'est pas terminé, d'où le *presque*.
7. D'où le fait que le type d'une référence est `ref type` est non `type`.

OCaml
```
for i = 0 to 99 do
  expr
done
```

Lemorceau de code ci-dessus **est une expression**. Elle consiste à évaluer `expr` pour `i` allant de `0` **inclus** à `99` **inclus** ; afin d'appliquer ses effets secondaires. Ensuite, la valeur de la boucle `for` est `()` .

⚠ J'insiste : une boucle `for` est une expression de type `unit` !!
Cela a deux conséquences :

14. Si l'expression dans la boucle `for` n'a pas d'effets secondaires, la boucle ne sert à rien. Tout ce qu'elle ferait alors est calculer longuement... `()`.

15. La boucle doit être traitée comme une expression de type `unit`. Elle ne peut pas apparaitre n'importe où ou n'importe quand.

Voici une phrase OCaml qui affiche les entiers de 0 à 99 :

OCaml
```
let _ =
  for i = 0 to 99 do
    let _ = print_int i in
    print_char '\n'
  done
```

16. Vérifiez que vous comprenez cette phrase.

17. Testez-la.

Vous noterez que pour accéder au compteur de boucle, il suffit de l'appeler comme on appelerait n'importe quelle variable. Le compteur n'est pas une référence, il n'est pas mutable ; c'est comme-ci il était recréé à chaque tour de boucle.
Voici une fonction OCaml qui double le contenu d'un tableau, écrite avec une boucle `for` :

OCaml
```
let double\_contenu = fun t ->
  for i = 0 to Array.length t -1 do
    t.(i) <- 2* t.(i)
  done
```

18. Pourquoi la boucle va-t-elle jusqu'à `Array.length t -1` ?

19. Quel est le type de la fonction ? Quels « indices » le compilateur utilise-t-il pour calculer ce type ?

20. Tester la fonction pour vérifier qu'elle fonctionne bien.

Un peu d'entrainement :

21. Écrire une fonction qui tri un tableau.
Je recommande de faire un tri bulle, cf DM des vacances d'octobre. C'est l'un des plus simples à implémenter - ce qui ne veut pas dire que ce sera facile. N'hésitez pas à demander de l'aide.

Combinonons maintenant références, tableaux et boucles.

22. Que fait la fonction ci-dessous ? Le vérifier.

OCaml
```
let surprise = fun t ->
  let m = ref t.(0) in
  let _ =
    for i = 0 to Array.length t -1 do
      if t.(i) < !m then m := t.(i)
    done in
  !m
```

23. Écrire une fonction qui prend en argument un tableau (dont on suppose qu'il a au moins deux cases) et renvoie la valeur du plus grand écart entre deux cases consécutives.
Autrement dit, on cherche le maximum entre `|t.(1)-t.(0)|`, `|t.(2)-t.(1)|`, etc. Attention, la valeur absolue n'existe pas en OCaml, il faudra la recoder.

C.2 Tant Que

La grande soeur des boucles Pour existe aussi, bien sûr. Il s'agit aussi d'une **expression de type `unit`**.
Voici la syntaxe de **l'expression** boucle while :

OCaml
```
while booleen do
  expr
done
```

Cette **expression de type unit** [8] va répéter `expr` tant que `booleen` est vrai.

Voici par exemple une fonction qui doit renvoyer l'indice de la première case d'un tableau d'entier qui contient un entier strictement supérieur à 10 :

OCaml
```
let demo = fun t ->
  let i = ref 0 in
  let _ =
    while t.(!i) <= 10 do
      i := !i +1
    done in
  !i
```

24. La fonction précédente fonctionne s'il existe un entier strictement supérieur à 10. Si il n'y en a pas, elle provoque une erreur. Pourquoi ?

Corriger l'erreur. On renverra `-1` s'il n'y a pas de tel entier.

D Tips and tricks

D.1 L'opérateur `;`

Tous ces `let _ = ... in ...` pour manipuler des expressions de type `unit` sont épuisants. Pour les éviter, on peut utiliser l'opérateur `;` : `e0 ; e1` signifie « évaluer `e0` et appliquer ses effets secondaires, jeter sa valeur à la poubelle, puis évaluer `e1` et appliquer ses effets secondaires et s'évaluer en sa valeur ».

Par exemple, la fonction d'affichage se réécrit :

OCaml
```
let demo = fun t ->
  let i = ref 0 in
  while t.(!i) <= 10 do
    i := !i +1
  done;
  !i
```

Il y a un point sur lequel il faut cependant faire très attention avec cet opérateur : sa priorité n'est pas ce que l'on pense. Il est *moins* prioritaire que *if then else*, et cela est embetant.

Ainsi, l'expression `if b then e0; e1 else e2` est parenthésée comme :
`(if b then e0); (e1 else e2)`. Ce qui n'a aucun sens et ne compile donc pas (le `else` ne correspond à aucun `if` ici...).

La solution est de « forcer » le bon parenthésage : `if b then (e0; e1) else e2`. On voudrait toutefois conserver les parenthèses pour « les maths ». Aussi, lorsque l'on veut utiliser des parenthèses pour encadrer une suite d'expressions séparées par des `;`, on utilise `begin` (qui est parenthèse ouvrante) et `end` (qui est une parenthèse fermante).

25. Tester le code `let x = 3 * begin 1 + 2 end` . Vérifier que `begin` et `end` se comportent bien comme des parenthèses « normales ».

26. Écrire une fonction qui affiche le contenu d'un tableau d'entiers. On séparera les cases par des virgules, et on fera un retour à la ligne à la fin.

27. Réécrire toutes vos fonctions depuis le début du TP en utilisant `;` quand nécessaire.

⚠Une erreur usuelle est de vouloir appliquer `;` à des phrases. Cela ne fonctionne pas. `;` s'applique à des *expressions*.

D.2 Module Array

Les fonctions de manipulation de tableaux que nous avons vues jusqu'à présent commençaient toutes par `Array.` : en fait, ce préfixe signifie « aller chercher dans la librairie Array la fonction ». En OCaml, on préfère parler de modules que de librairies [9].

8. Non, je n'insiste pas trop. Croyez-moi.
9. Mais c'est pareil.

Vous pouvez trouver la documentation de ce module à l'aide de la commande de terminal `man Array`. Elle est en ligne ici : `https://v2.ocaml.org/api/Array.html`.

28. (Évaluation transversale de l'anglais) Que fait la fonction `get` ? Et la fonction `iter` ?

29. À l'aide de `iter` , réécrire votre fonction d'affichage d etableua d'entier *sans utiliser de boucle ni de fonction récursive.*

30. (Bonus) À l'aide de `fold_left` , écrire une fonction qui calcule la somme d'un tableau d'entiers.

E Pour aller plus loin

31. Écrire les fonctions suivantes (dont le but est de vous faire manipuler des récursions/boucles/tableaux) :

- **a.** Tri rapide (‼).
- **b.** Multiplication de matrices (on représentera une matrice par un tableau de tableaux).
- **c.** Pivot de Gauss (idem).